12 | 打开首页之二：如何平衡利用硬件资源?

2021-04-16 高楼

高楼的性能工程实战课 进入课程 >

讲述：高楼

时长 21:56 大小 20.10M

你好，我是高楼。

针对打开首页接口的性能问题，我们在上节课中确定了是 Gateway 在消耗响应时间，达到了近 100 毫秒。于是，我们开始定位 Gateway 上的响应时间消耗。

在第一阶段的时候，我们关注了应用所在的主机，同时还了解到，宿主机总共有四台机器；在第二阶段，我们查看了物理机的 CPU 模式。并尝试通过修改 CPU 运行模式来优化性能。可是，问题仍然没有解决，TPS 没见提升，响应时间依旧很长。

今天这节课，我们进入第三阶段，继续分析其他的瓶颈点，比如 wa cpu、资源均衡使用、网络带宽等问题。其中，**在性能的分析逻辑里，资源均衡使用是一个非常容易被忽略，但**

又极为重要的方面。我们通常都盯着计数器给出的数值有什么异常，而不是考虑资源怎么做相应的调配。

在我们这个案例中，系统是用 k8s 来管理资源的，所以我们必须要关注资源的均衡使用，避免出现有些服务性能很差，却和性能好的服务分配同样资源的情况。另外，网络资源在 k8s 中会跨越好几层，我们也要着重关注一下。

在学习这节课时，我建议你多思考下资源的均衡使用问题。现在，我们就开始今天的课程。

定位 gateway 上的响应时间消耗

第三阶段：NFS 服务器的 wa cpu 偏高

根据分析的逻辑，我们仍然是先看全局监控数据，思路依旧是“全局 - 定向”，这是我一贯的顺序了。

因此，我们现在再来查一下全局监控计数器，得到下面这样的视图：

复制代码

```
[root@lenvo-nfs-server ~]# top
top - 00:12:28 up 32 days,  4:22,  3 users,  load average: 9.89, 7.87, 4.71
Tasks: 217 total,   1 running, 216 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.0 us,  4.0 sy,  0.0 ni, 34.8 id, 61.2 wa,  0.0 hi,  0.0 si,  0.0 s
%Cpu1  :  0.0 us,  4.7 sy,  0.0 ni, 27.8 id, 67.6 wa,  0.0 hi,  0.0 si,  0.0 s
%Cpu2  :  0.0 us,  6.1 sy,  0.0 ni,  0.0 id, 93.9 wa,  0.0 hi,  0.0 si,  0.0 s
%Cpu3  :  0.0 us,  7.6 sy,  0.0 ni,  3.4 id, 82.8 wa,  0.0 hi,  6.2 si,  0.0 s
KiB Mem :  3589572 total,    82288 free,   775472 used,  2731812 buff/cache
KiB Swap:  8388604 total,  8036400 free,   352204 used.  2282192 avail Mem
```

可以看到，计数器 wa 的 CPU 使用率偏高，其中 Cpu2 的 wa 已经达到 90% 以上。我们知道，wa cpu 是指 CPU 在读写的时候，所产生的 IO 等待时间占 CPU 时间的百分比。那么，它现在竟然这么高，是因为写操作有很多吗？

这时候我们就要关注下 IO 的状态了，因为 IO 慢绝对是一个性能问题。通过 iostat 命令，我们看到 IO 状态如下：

```
[root@lenvo-nfs-server ~]# iostat -x -d 1
Linux 3.10.0-693.el7.x86_64 (lenvo-nfs-server)   2020年12月26日   _x86_64_  (4
..................
Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avg
sda               0.00     0.00   94.00   39.00 13444.00 19968.00   502.44   1


Device:         rrqm/s   wrqm/s     r/s     w/s    rkB/s    wkB/s avgrq-sz avg
sda               0.00    18.00  137.00  173.00 17712.00 43056.00   392.05   1
..................
```

你可以看到，IO 使用率达到了 100%，说明 IO 的过程实在是太慢了。

接下来，我们再查查 Block Size 是多少，算一下当前 IO 到底是随机读写还是顺序读写。虽然大部分操作系统都默认 Block Size 是 4096，但是，本着不出小错的原则，我们还是查一下比较放心。

我们先确定磁盘的格式是什么：

```
[root@lenvo-nfs-server ~]# cat /proc/mounts
..................
/dev/sda5 / xfs rw,relatime,attr2,inode64,noquota 0 0
..................
[root@lenvo-nfs-server ~]#
```

通过上述命令可以知道，这个磁盘是 XFS 格式。那我们就用下面这个命令来查看 Block Size：

```
[root@lenvo-nfs-server ~]# xfs_info /dev/sda5
meta-data=/dev/sda5              isize=512    agcount=4, agsize=18991936 blks
         =                       sectsz=512   attr=2, projid32bit=1
         =                       crc=1        finobt=0 spinodes=0
data     =                       bsize=4096   blocks=75967744, imaxpct=25
         =                       sunit=0      swidth=0 blks
naming   =version 2              bsize=4096   ascii-ci=0 ftype=1
log      =internal               bsize=4096   blocks=37093, version=2
         =                       sectsz=512   sunit=0 blks, lazy-count=1
realtime =none                   extsz=4096   blocks=0, rtextents=0
[root@lenvo-nfs-server ~]#
```

结果显示，Block Size 是 4096。同时，我们也可以看到读写基本上都是顺序的，不是随机。

那我们就来计算一条数据，确认一下顺序写的能力。如果全部是随机写，那么：

$$次数 = （43056 \times 1024） \div 4096 = 10,764次$$

但是，实际上写只有 173 次，所以确实是顺序写了。

问题又来了，一次写多少个 Block 呢？

$$（43056 \times 1024） \div 173 \div 4096 \approx 62个$$

我们得出，一次写 62 个 Block。从这样的数据来看，说明顺序写的能力还是不错的。因为对普通磁盘来说，应用在读写的时候，如果是随机写多，那写的速度就会明显比较慢；如果顺序写多，那么写的速度就可以快起来。

你发现了吗？虽然当前磁盘的顺序写能力不错，但是等待的时间也明显比较多。所以，接下来，我们得查一下是什么程序写的。这里我们用 iotop 命令查看：

复制代码

```
Total DISK READ :      20.30 M/s | Total DISK WRITE :     24.95 M/s
Actual DISK READ:      20.30 M/s | Actual DISK WRITE:       8.27 M/s
  TID  PRIO  USER     DISK READ  DISK WRITE  SWAPIN     IO>    COMMAND
12180 be/4 root        2.39 M/s   16.01 M/s  0.00 % 35.94 % [nfsd]
12176 be/4 root        3.20 M/s    0.00 B/s  0.00 % 32.25 % [nfsd]
12179 be/4 root        3.03 M/s    6.43 M/s  0.00 % 32.23 % [nfsd]
12177 be/4 root        2.44 M/s  625.49 K/s  0.00 % 31.64 % [nfsd]
12178 be/4 root        2.34 M/s 1473.47 K/s  0.00 % 30.43 % [nfsd]
12174 be/4 root        2.14 M/s   72.84 K/s  0.00 % 29.90 % [nfsd]
12173 be/4 root        2.91 M/s  121.93 K/s  0.00 % 24.95 % [nfsd]
12175 be/4 root     1894.69 K/s   27.71 K/s  0.00 % 24.94 % [nfsd]
.............
```

可以看到，IO 都是 NFS 写过来的。那 NFS 的流量又是从哪里来的呢？从下面的数据来看，这些流量是从各个挂载了 NFS 盘的机器写过来的，这是我们一开始部署应用的时候，

考虑统一使用 NFS 来做 IO 的思路。因为这个机器挂载了一个大容量的磁盘，为了保证磁盘够用，就把多个主机挂载了 NFS 盘。

```
                     191Mb               381Mb               572Mb
mqqqqqqqqqqqqqqqqqqqvqqqqqqqqqqqqqqqqqqqvqqqqqqqqqqqqqqqqqqqvqqqqqqqqqqqqqqqqqq
172.16.106.119:nfs                   => 172.16.106.130:multiling-http        1
                                     <=                                      2
172.16.106.119:nfs                   => 172.16.106.100:apex-mesh             1
                                     <=                                      2
172.16.106.119:nfs                   => 172.16.106.195:vatp
                                     <=                                      9
172.16.106.119:nfs                   => 172.16.106.56:815                    7
                                     <=
172.16.106.119:nfs                   => 172.16.106.79:device                 1
                                     <=                                      1
172.16.106.119:ssh                   => 172.16.100.201:cnrprotocol           2
                                     <=
169.254.3.2:60010                    => 225.4.0.2:59004                      2
                                     <=
169.254.6.2:60172                    => 225.4.0.2:59004                      2
                                     <=
172.16.106.119:nfs                   => 172.16.106.149:986
                                     <=


qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
TX:             cum:   37.0MB   peak:   31.9Mb                    rates:   3
RX:                     188MB            106Mb                            6
TOTAL:                  225MB            111Mb                            6
```

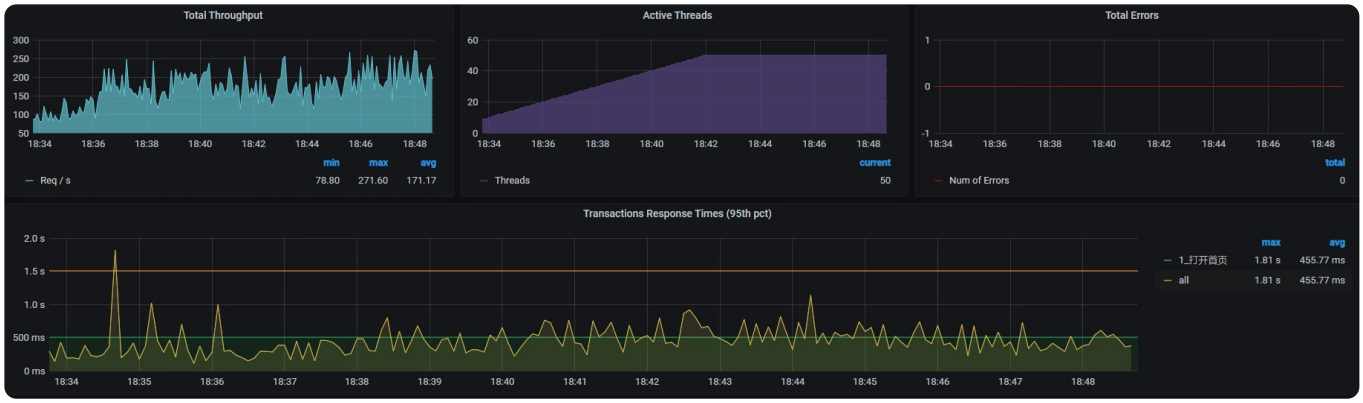
我们在 Total DISK WRITE 中可以看到，读能力才达到 20M。没办法，既然 wa 这个机器的能力不怎么好，那就只有放弃统一写的思路。不过，为了不让机器的 IO 能力差成为应用的瓶颈点，我们还是再尝试一下这两个动作：

第一，把 MySQL 的数据文件移走；

第二，把 Log 移走。

接着我们执行场景，希望结果能好。

可是，在我查看了 TPS 和 RT 曲线后，很遗憾地发现，结果并没有改善。TPS 依然很低并且动荡非常大：



看来我们的努力并没有什么效果，悲剧！就这样，命运让我们不得不来到第四个阶段。

第四阶段：硬件资源耗尽，但 TPS 仍然很低

这个阶段我们查什么呢？仍然是全局监控的数据。我们来看一下所有主机的 Overview 资源：

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	1.13 week	7.64 GiB	4	0.66	12.67%	30.03%	29.95%	0 B/s	95.70 kB/s	307	24	1.36 Mbps	1.36 Mbps
k8s-master-2	k8s-master-2	1.13 week	7.64 GiB	4	0.93	23.99%	26.96%	29.95%	0 B/s	92.39 kB/s	274	23	254.49 kbps	354.99 kbps
k8s-master-3	k8s-master-3	1.13 week	7.64 GiB	4	0.81	9.86%	27.73%	29.95%	0 B/s	93.75 kB/s	286	29	329.04 kbps	513.59 kbps
k8s-worker-1	k8s-worker-1	7.18 hour	15.51 GiB	6	2.88	29.87%	54.79%	32.21%	18.57 kB/s	85.36 kB/s	31	11 K	22.08 Mbps	43.27 Mbps
k8s-worker-2	k8s-worker-2	6.98 hour	15.51 GiB	6	0.37	7.80%	43.43%	29.95%	13.81 kB/s	3.92 kB/s	29	93	10.98 Mbps	11.64 Mbps
k8s-worker-3	k8s-worker-3	7.21 hour	15.51 GiB	6	0.21	5.81%	24.54%	29.95%	18.41 kB/s	5.96 kB/s	30	43	91.84 kbps	130.20 kbps
k8s-worker-4	k8s-worker-4	7.17 hour	15.51 GiB	6	0.52	9.78%	19.92%	29.95%	18.41 kB/s	30.84 kB/s	43	167	852.98 kbps	1.18 Mbps
k8s-worker-5	k8s-worker-5	6.71 hour	15.51 GiB	6	4.37	24.19%	36.75%	33.36%	4.60 kB/s	2.19 kB/s	30	73	5.68 Mbps	8.74 Mbps
k8s-worker-6	k8s-worker-6	6.65 hour	15.51 GiB	6	0.90	16.45%	38.38%	35.18%	13.81 kB/s	5.23 kB/s	30	30	839.37 kbps	749.37 kbps
k8s-worker-7	k8s-worker-7	6.23 hour	15.51 GiB	4	6.22	39.51%	33.15%	36.50%	4.60 kB/s	6.38 kB/s	29	153	3.10 Mbps	2.02 Mbps
k8s-worker-8	k8s-worker-8	6.23 hour	15.51 GiB	4	33.86	95.95%	44.61%	41.64%	18.41 kB/s	17.66 kB/s	36	12 K	23.90 Mbps	23.13 Mbps
k8s-worker-9	k8s-worker-9	6.23 hour	15.51 GiB	6	2.08	22.83%	42.88%	29.95%	18.41 kB/s	9.52 kB/s	30	25	5.04 Mbps	9.68 Mbps

从上图中可以看到，虚拟机 k8s-worker-8 的 CPU 使用率已经很高了，达到了 95.95%。那我们就登录到这台虚拟机上，看看更详细的全局监控数据：

```
top - 18:48:30 up  6:12,  2 users,  load average: 35.37, 33.30, 25.99
Tasks: 294 total,   2 running, 292 sleeping,   0 stopped,   0 zombie
%Cpu0  : 69.6 us, 12.6 sy,  0.0 ni,  4.9 id,  0.0 wa,  0.0 hi, 12.6 si,  0.3 st
%Cpu1  : 70.4 us, 13.7 sy,  0.0 ni,  1.7 id,  0.0 wa,  0.0 hi, 13.7 si,  0.3 st
%Cpu2  : 70.7 us, 12.7 sy,  0.0 ni,  5.3 id,  0.0 wa,  0.0 hi, 11.0 si,  0.4 st
%Cpu3  : 70.1 us, 13.7 sy,  0.0 ni,  4.6 id,  0.0 wa,  0.0 hi, 11.3 si,  0.4 st
KiB Mem : 16266296 total,  4871492 free,  6849724 used,  4545080 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  9027236 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
31199 root      20   0 7732384 492784  15752 S 147.5  3.0  25:33.43 java -Dapp.id=svc-mall-portal -javaagent:/opt/skywalking
30074 root      20   0 7735376 476244  15708 S  91.1  2.9  17:55.90 java -Dapp.id=svc-mall-gateway -javaagent:/opt/skywalkin
18934 root      20   0 3713524 998.9m  18748 S  38.9  6.3  52:10.08 java -XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemo
 3657 root      20   0 1504888 155740  33128 R  23.1  1.0  64:11.40 /home/weave/scope --mode=probe --probe-only --probe.kube
 1069 root      20   0 1259824 114000  29728 S  15.8  0.7  59:16.96 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/c
 1062 root      20   0 2263836  61220  16220 S  13.5  0.4  53:35.08 /usr/bin/containerd
 1059 root      20   0 2575920 106084  37988 S  12.2  0.7  51:04.26 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/
15018 root      20   0 5919448   1.3g  16492 S   3.3  8.5  25:54.74 /usr/lib/jvm/java-1.8.0-openjdk/bin/java -Xms2g -Xmx2g -
26756 root      20   0 7730320 461712  15692 S   3.0  2.8   4:48.35 java -Dapp.id=svc-mall-member -javaagent:/opt/skywalking
28451 root      20   0 7732356 451080  15728 S   3.0  2.8   4:28.37 java -Dapp.id=svc-mall-admin -javaagent:/opt/skywalking/
 4723 root      20   0 1484184  42336  17436 S   2.6  0.3  17:23.77 calico-node -felix
 9293 root      20   0 2808944 117108  77740 S   2.0  0.7   9:13.75 /usr/lib64/firefox/firefox -contentproc -childID 4 -isFo
27683 root      20   0 7698508 438196  15680 S   2.0  2.7   3:39.70 java -Dapp.id=svc-mall-demo -javaagent:/opt/skywalking/a
 3756 nfsnobo+  20   0  115724  17996   5508 S   1.7  0.1   3:38.25 /bin/node_exporter --web.listen-address=127.0.0.1:9100 -
 8591 root      20   0 3071800 265012  89540 S   1.3  1.6   9:13.24 /usr/lib64/firefox/firefox
32357 root      20   0 7719976 378044  15668 S   1.3  2.3   3:25.88 java -Dapp.id=svc-mall-monitor -javaagent:/opt/skywalkin
```

因为 CPU 不超分了，所以我们可以很明显地看到，k8s-worker-8 中的 CPU 被耗尽。从进程上来看，CPU 是被我们当前正在测试的接口服务消耗的。并且在这台虚拟机上，不止有 Portal 这一个进程，还有很多其他的服务。

那我们就把 Portal 服务调度到一个不忙的 worker 上去，比如移到 worker-3（6C16G）上：

```
svc-mall-portal-5d46bf646-cq2bc   1/1   Running       0   20s   10.100.69.227   k8s-worker-3   <none>   <none>
svc-mall-portal-74778bc78b-jrrh6  0/1   Terminating   0   29m   <none>          k8s-worker-8   <none>   <none>
```

得到如下结果：

我们看到，TPS 已经有所上升了，达到了近 300，性能确实变好了一些。但是，这个数据还不如我们一开始不优化的结果，毕竟一开始还能达到 300TPS 呢。那我们就接着分析当前的瓶颈在哪里。

我们先来看一下主机的性能数据：

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	1.13 week	7.64 GiB	4	0.84	12.73%	29.80%	29.95%	0 B/s	92.64 kB/s	307	20	1.16 Mbps	1.16 Mbps
k8s-master-2	k8s-master-2	1.13 week	7.64 GiB	4	1.51	24.06%	27.00%	29.95%	0 B/s	92.90 kB/s	273	25	254.08 kbps	353.69 kbps
k8s-master-3	k8s-master-3	1.13 week	7.64 GiB	4	2.32	10.58%	27.61%	29.95%	0 B/s	94.03 kB/s	285	25	323.20 kbps	511.25 kbps
k8s-worker-1	k8s-worker-1	7.42 hour	15.51 GiB	6	2.97	34.91%	55.02%	32.08%	36.82 kB/s	74.54 kB/s	30	14 K	28.65 Mbps	55.79 Mbps
k8s-worker-2	k8s-worker-2	7.21 hour	15.51 GiB	6	0.34	8.11%	43.37%	29.95%	18.41 kB/s	9.22 kB/s	29	100	12.55 Mbps	13.23 Mbps
k8s-worker-3	k8s-worker-3	7.44 hour	15.51 GiB	6	2.87	25.83%	27.94%	29.95%	9.21 kB/s	4.93 kB/s	29	222	29.07 Mbps	28.10 Mbps
k8s-worker-4	k8s-worker-4	7.40 hour	15.51 GiB	6	0.89	9.78%	19.93%	29.95%	9.21 kB/s	12.24 kB/s	42	184	853.65 kbps	1.18 Mbps
k8s-worker-5	k8s-worker-5	6.94 hour	15.51 GiB	6	3.20	26.06%	36.72%	33.37%	10.97 kB/s	2.90 kB/s	30	67	6.80 Mbps	14.58 Mbps
k8s-worker-6	k8s-worker-6	6.88 hour	15.51 GiB	6	0.67	16.43%	38.46%	35.19%	23.02 kB/s	7.79 kB/s	30	36	787.91 kbps	703.01 kbps
k8s-worker-7	k8s-worker-7	6.46 hour	15.51 GiB	4	6.64	53.18%	33.09%	36.50%	13.81 kB/s	8.60 kB/s	29	151	3.70 Mbps	2.69 Mbps
k8s-worker-8	k8s-worker-8	6.46 hour	15.51 GiB	4	23.12	90.12%	42.11%	41.59%	200.82 kB/s	466.28 kB/s	36	15 K	20.65 Mbps	26.67 Mbps
k8s-worker-9	k8s-worker-9	6.46 hour	15.51 GiB	6	2.14	24.06%	42.92%	29.95%	20.25 kB/s	5.82 kB/s	29	25	5.74 Mbps	11.28 Mbps

其中，worker-8 的 CPU 使用率达到了 90.12%。为什么这个 CPU 还是如此之高呢？我们继续来 top 一下，看看 worker-8 的性能数据：

```
top - 19:22:56 up  6:47,  3 users,  load average: 27.87, 26.74, 23.81
Tasks: 294 total,   3 running, 291 sleeping,   0 stopped,   0 zombie
%Cpu0  : 64.9 us, 17.2 sy,  0.0 ni,  2.7 id,  0.0 wa,  0.0 hi, 15.2 si,  0.0 st
%Cpu1  : 64.9 us, 13.5 sy,  0.0 ni,  5.1 id,  0.0 wa,  0.0 hi, 16.2 si,  0.3 st
%Cpu2  : 59.6 us, 13.7 sy,  0.0 ni, 13.0 id,  0.0 wa,  0.0 hi, 13.3 si,  0.4 st
%Cpu3  : 60.3 us, 16.2 sy,  0.0 ni,  9.7 id,  0.0 wa,  0.0 hi, 13.4 si,  0.3 st
KiB Mem : 16266296 total,  5193600 free,  6496156 used,  4576540 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  9374052 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
30074 root      20   0 7735376 554416  15740 S 172.6  3.4  62:30.70 java -Dapp.id=svc-mall-gateway -javaagent:/opt/skywa
18934 root      20   0 3712496   1.0g  18748 S  65.0  6.4  68:52.70 java -XX:+UnlockExperimentalVMOptions -XX:+UseCGroup
 3657 root      20   0 1572924 146176  33136 S  41.3  0.9  75:54.76 /home/weave/scope --mode=probe --probe-only --probe.
 1069 root      20   0 1259824 112812  29728 S  18.2  0.7  64:15.63 /usr/bin/dockerd -H fd:// --containerd=/run/containe
 1062 root      20   0 2263836  61912  16224 S  14.5  0.4  57:54.31 /usr/bin/containerd
 1059 root      20   0 2575920 105532  38012 S  11.6  0.6  55:45.83 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kuberne
26756 root      20   0 7730320 468372  15700 S   6.3  2.9   6:35.64 java -Dapp.id=svc-mall-member -javaagent:/opt/skywal
15018 root      20   0 5919448   1.3g  16492 S   5.6  8.5  27:42.76 /usr/lib/jvm/java-1.8.0-openjdk/bin/java -Xms2g -Xmx
28451 root      20   0 7733384 455528  15728 S   4.3  2.8   6:12.12 java -Dapp.id=svc-mall-admin -javaagent:/opt/skywalk
```

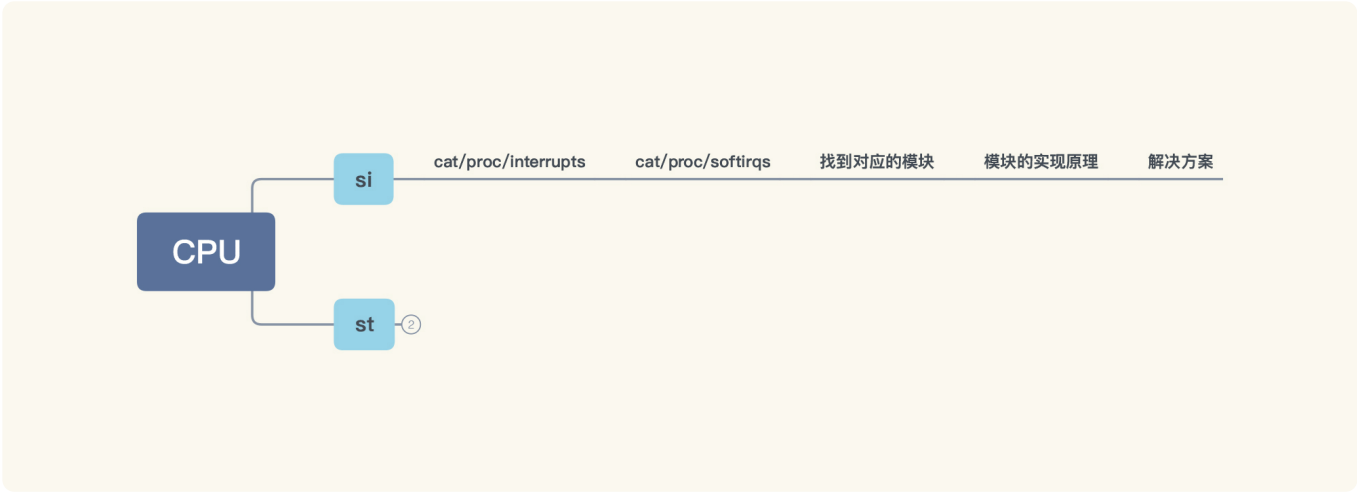
你看，在 process table 中，排在最上面的是 Gateway 服务，说明是 Gateway 进程消耗的 CPU 最多。既然如此，我们自然要看看这个进程中的线程是不是都在干活。

我们看到上图中全是绿色的，也就是说 Gateway 中的线程一直处于 Runnable 状态，看来工作线程确实挺忙的了。而在前面的 worker-8 性能数据中，si cpu 已经达到了 16% 左右。所以结合这一点，我们来看一下实时的软中断数据：

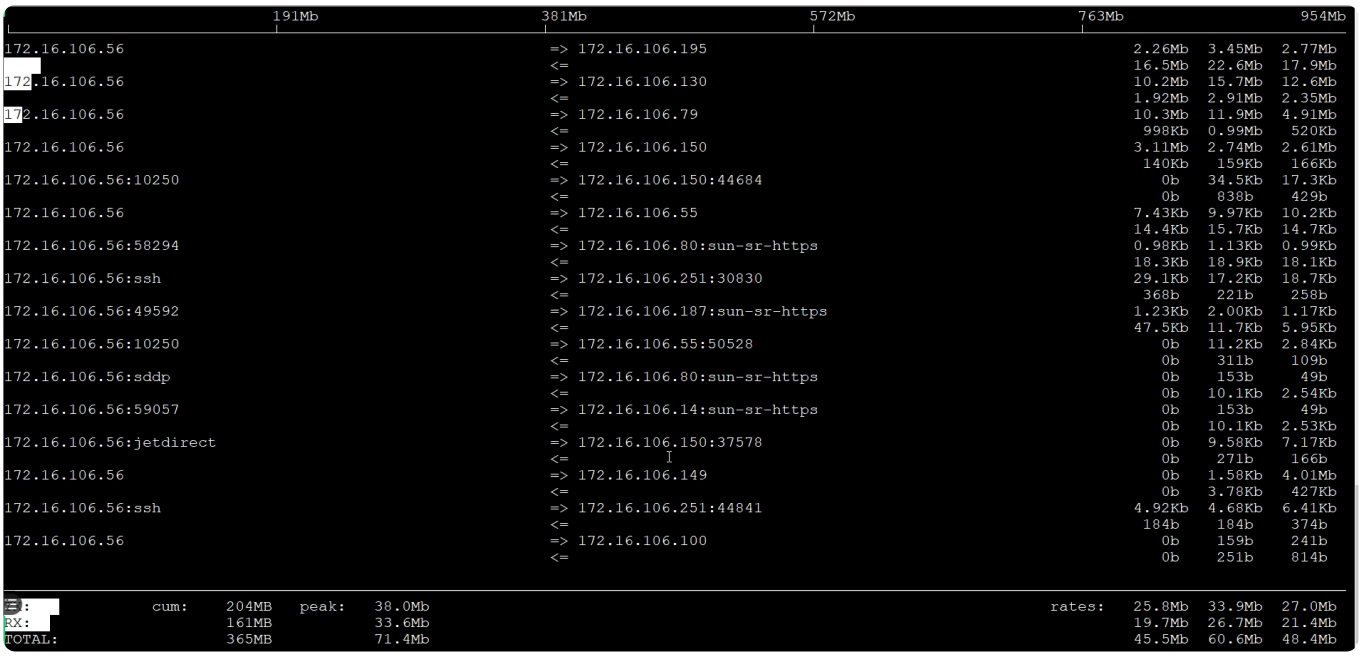
```
Every 1.0s: cat /proc/softirqs

                    CPU0       CPU1       CPU2       CPU3
          HI:          3          2          3          3
       TIMER:    15505874   15644095   15629307   15583182
      NET_TX:          26         22         36         48
      NET_RX:    19551675   19855797   19866368   19919765
       BLOCK:       32503      35244      38880      36421
BLOCK_IOPOLL:           0          0          0          0
     TASKLET:        1922       1884       1760       1912
       SCHED:     4264422    3871472    3570983    3434464
     HRTIMER:           0          0          0          0
         RCU:     8805028    8839328    8828550    8797573
```

可以看到网络软中断一直在往上跳，这说明确实是网络软中断导致 si cpu 变高的。网络软中断的变化是我们根据证据链找下来的。证据链如下：



我们再看一下网络带宽有多大：



可以看到，网络带宽倒是不大。

从上述 Gateway 的工作线程、软中断数据和网络带宽情况来看，Gateway 只负责转发，并没有什么业务逻辑，也没有什么限制。所以，针对 TPS 上不去的原因，似乎除了网络转发能力比较差之外，我们再找不到其他解释了。

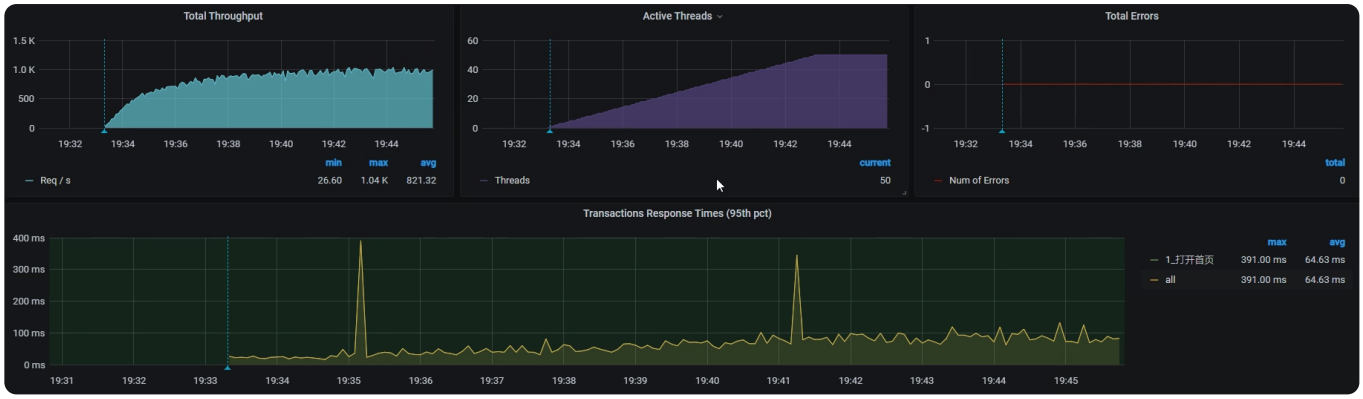
这个思路其实是需要一些背景知识的，因为我们通常用网络带宽来判断网络是不是够用，但是这是不够的。你要知道，在网络中当小包过多的时候，网络带宽是难以达到线性流量的。所以，我们这里的网络带宽即便不会很高，也会导致网络软中断的增加和队列的出现。

既然如此，那我们就把这个 Gateway 也从 worker-8 移到 worker-2（6C16G）上去，做这一步是为了减少网络软中断的争用。我们再看一下集群的整体性能：

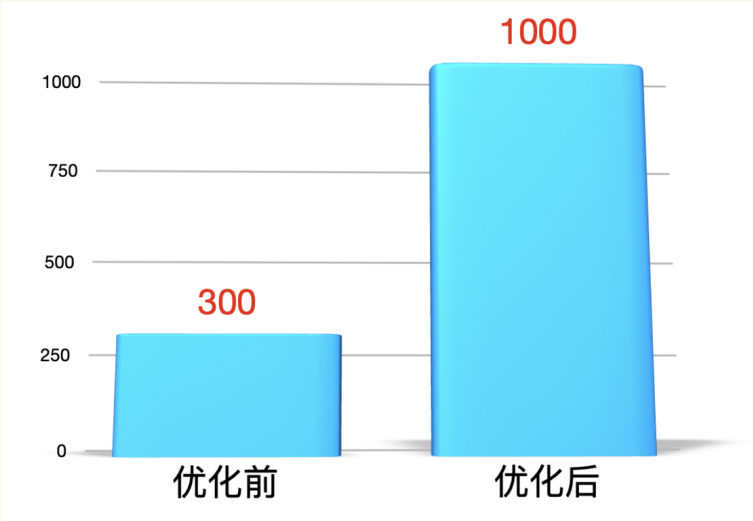
IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	1.14 week	7.64 GiB	4	1.10	14.97%	29.86%	29.95%	0 B/s	90.07 kB/s	308	27	1.16 Mbps	1.16 Mbps
k8s-master-2	k8s-master-2	1.14 week	7.64 GiB	4	1.19	24.05%	26.87%	29.95%	0 B/s	89.71 kB/s	273	20	250.57 kbps	359.33 kbps
k8s-master-3	k8s-master-3	1.14 week	7.64 GiB	4	0.41	11.87%	27.54%	29.95%	0 B/s	91.43 kB/s	284	27	315.81 kbps	493.01 kbps
k8s-worker-1	k8s-worker-1	8.17 hour	15.51 GiB	6	9.10	66.19%	56.37%	32.12%	23.02 kB/s	271.25 kB/s	30	11 K	89.15 Mbps	155.28 Mbps
k8s-worker-2	k8s-worker-2	7.96 hour	15.51 GiB	6	8.82	57.89%	47.90%	29.95%	32.22 kB/s	12.85 kB/s	29	22 K	55.21 Mbps	66.55 Mbps
k8s-worker-3	k8s-worker-3	8.19 hour	15.51 GiB	6	11.11	70.78%	28.19%	29.95%	13.81 kB/s	9.16 kB/s	29	574	99.22 Mbps	96.39 Mbps
k8s-worker-4	k8s-worker-4	8.16 hour	15.51 GiB	6	0.85	10.93%	20.01%	29.95%	4.60 kB/s	7.73 kB/s	42	172	852.97 kbps	1.18 Mbps
k8s-worker-5	k8s-worker-5	7.70 hour	15.51 GiB	6	2.85	25.44%	36.77%	33.37%	13.81 kB/s	3.51 kB/s	30	45	8.23 Mbps	12.74 Mbps
k8s-worker-6	k8s-worker-6	7.63 hour	15.51 GiB	6	2.12	17.78%	38.40%	35.19%	18.41 kB/s	6.12 kB/s	30	34	7.96 Mbps	7.90 Mbps
k8s-worker-7	k8s-worker-7	7.22 hour	15.51 GiB	4	2.72	43.60%	33.23%	36.51%	13.81 kB/s	8.69 kB/s	29	152	2.09 Mbps	1.18 Mbps
k8s-worker-8	k8s-worker-8	7.22 hour	15.51 GiB	4	5.34	56.65%	38.91%	41.53%	18.41 kB/s	14.34 kB/s	35	245	22.87 Mbps	21.82 Mbps
k8s-worker-9	k8s-worker-9	7.22 hour	15.51 GiB	6	1.66	27.28%	43.04%	29.95%	9.21 kB/s	8.30 kB/s	30	24	10.15 Mbps	19.51 Mbps

看起来不错哦，worker-3 的 CPU 使用率降到了 70.78%。不过网络带宽有几个地方变红了，这个我们后面再分析。至少我们从这里看到，压力是起来了。

我们回来看一下压力的情况：



TPS 已经达到 1000 左右了！棒棒的，有没有！我们画一个 TPS 对比图庆祝一下：

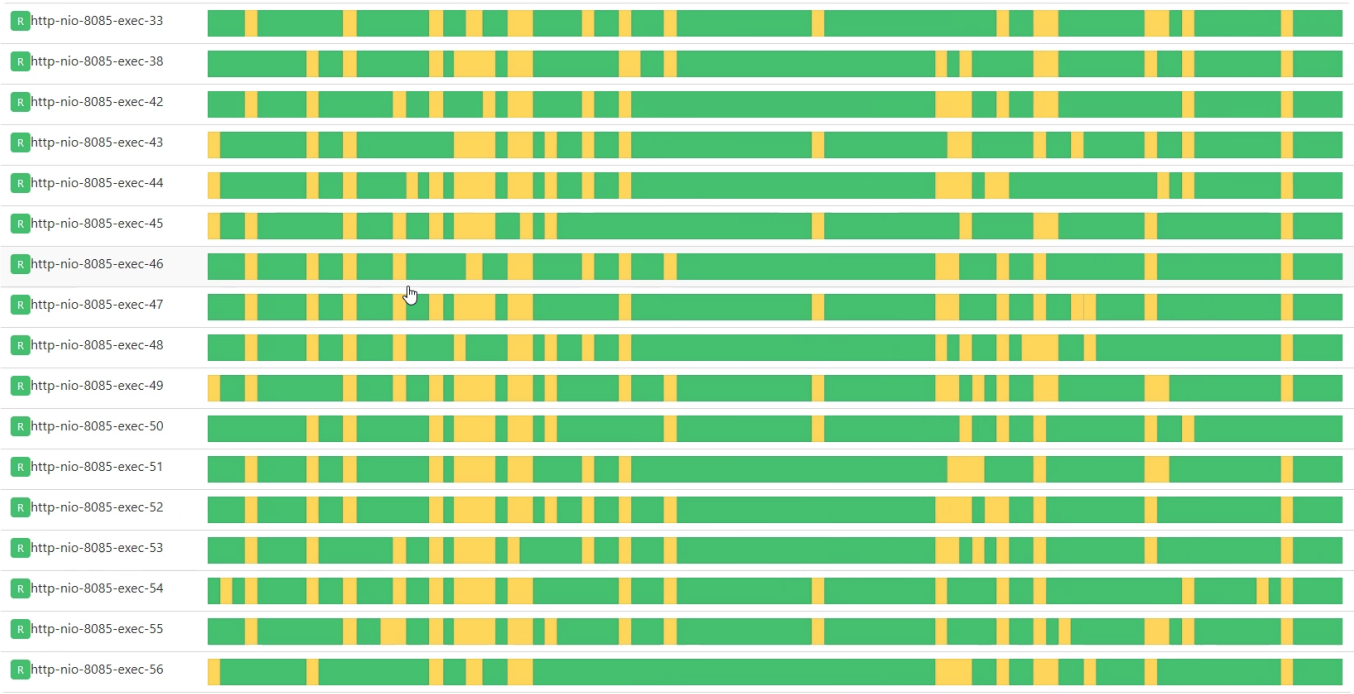


其实到这里，打开首页这个接口的基准场景就可以结束了，因为我们已经优化到了比需求还要高的程度。只是从技术角度来说，一个系统优化到最后是会有上限的，所以，我们仍然需要知道这个上限在哪里。

第五阶段：硬件资源还是要用完

现在压力把 worker-3 的 CPU 资源用得最高，用到了 70.78%。那么，下面我们就要把这个机器的硬件资源给用完，因为只有将资源都用尽，我们才能判断系统容量的最上限。这也就是我一直强调的，要将**性能优化分为两个阶段：一是把资源用起来；二是把容量调上去。**就算不是 CPU 资源，把其他的资源用完也可以。

既然这时候压力已经把 worker-3 的 CPU 资源用到了 70.78%，那我们就到这个应用中看一下线程把 CPU 用得怎么样。

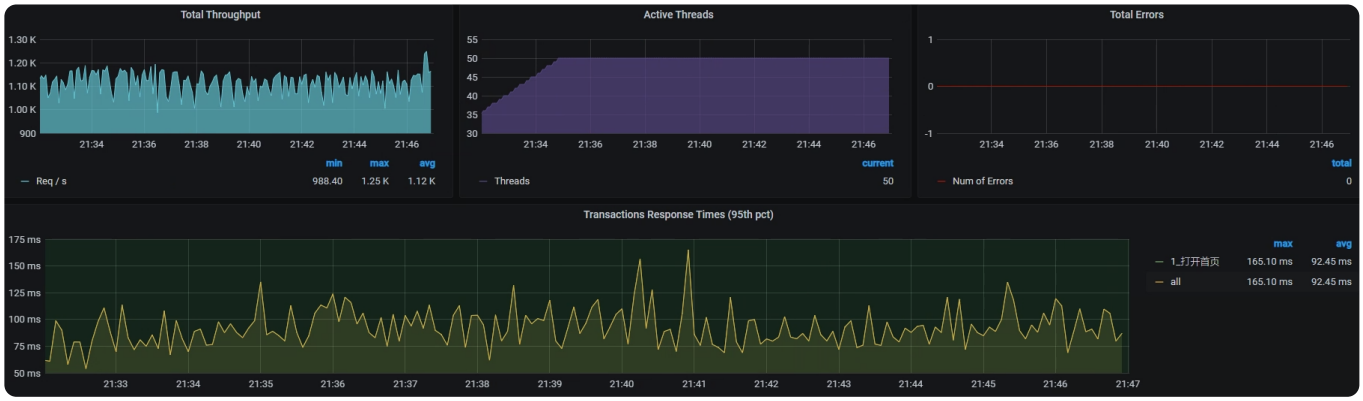


你看，这里面的线程确实都忙起来了。

既然如此，那我们把 Tomcat 和 JDBC 连接的最大值都改到 80，再来看一下 TPS 的表现（请你注意，这里只是一个尝试，所以改大即可，并没有什么道理。在后续的测试过程中，我们还要根据实际情况来做调整，就是不能让线程太大，也不能不够用）。

为了让压力能直接压到一个节点上，我们跳过 Ingress，用分段的测法直接把压力发到服务上。然后，我们去 Pod 里设置一个 node port 把服务代理出来，再修改一下压力脚本。

得到结果如下：

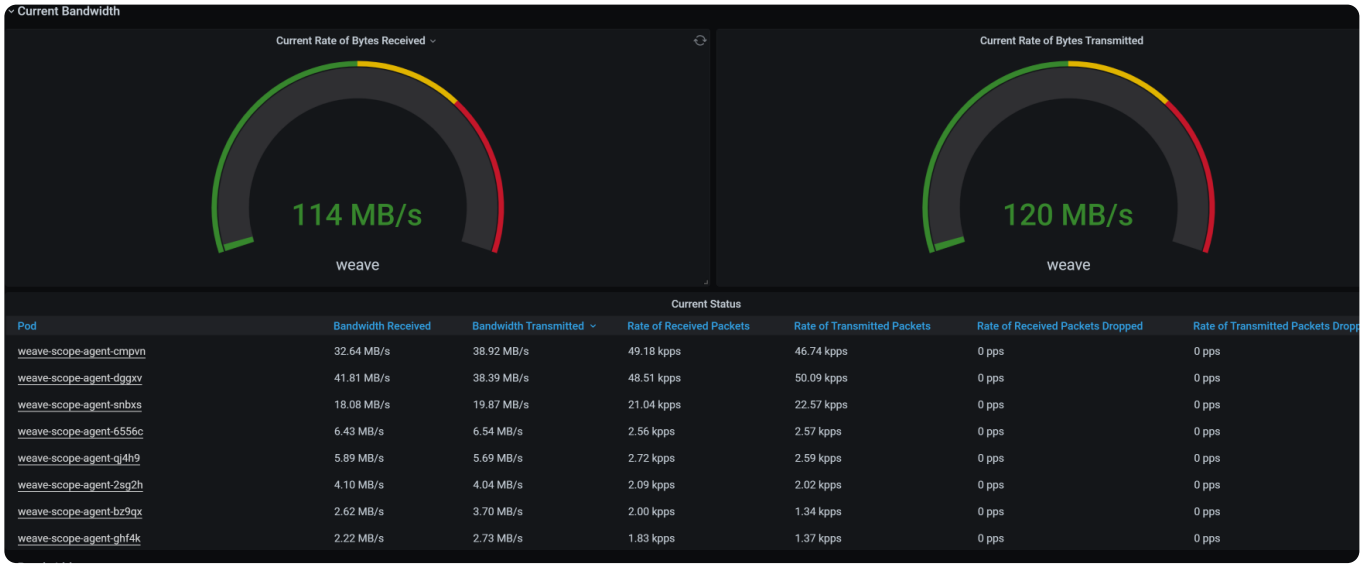


TPS 还是抖动大。那我们接着看全局监控：

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	1.15 week	7.64 GiB	4	0.72	14.41%	30.21%	29.95%	0 B/s	110.88 kB/s	309	26	1.17 Mbps	1.17 Mbps
k8s-master-2	k8s-master-2	1.15 week	7.64 GiB	4	1.65	23.88%	27.05%	29.95%	0 B/s	86.23 kB/s	272	23	237.70 kbps	327.17 kbps
k8s-master-3	k8s-master-3	1.15 week	7.64 GiB	4	0.73	11.75%	27.85%	29.95%	0 B/s	85.42 kB/s	284	27	316.23 kbps	473.64 kbps
k8s-worker-1	k8s-worker-1	10.61 hour	15.51 GiB	6	4.16	46.79%	55.34%	32.31%	90.50 kB/s	20.03 kB/s	31	203	106.14 Mbps	173.09 Mbps
k8s-worker-2	k8s-worker-2	10.40 hour	15.51 GiB	6	4.08	45.91%	47.68%	29.95%	13.81 kB/s	6.71 kB/s	29	210	60.34 Mbps	74.66 Mbps
k8s-worker-3	k8s-worker-3	10.63 hour	15.51 GiB	6	22.47	80.97%	28.65%	29.95%	16.11 kB/s	5.92 kB/s	33	1 K	116.44 Mbps	113.13 Mbps
k8s-worker-4	k8s-worker-4	10.59 hour	15.51 GiB	6	0.35	10.50%	20.13%	29.95%	18.41 kB/s	9.10 kB/s	42	180	850.96 kbps	1.19 Mbps
k8s-worker-5	k8s-worker-5	10.14 hour	15.51 GiB	6	3.13	25.27%	36.79%	33.39%	102.95 kB/s	2.39 kB/s	30	72	8.03 Mbps	12.11 Mbps
k8s-worker-6	k8s-worker-6	10.07 hour	15.51 GiB	6	1.29	16.87%	38.49%	35.20%	23.02 kB/s	25.77 kB/s	31	34	3.54 Mbps	3.31 Mbps
k8s-worker-7	k8s-worker-7	9.66 hour	15.51 GiB	4	9.62	56.59%	33.87%	36.51%	13.81 kB/s	9.58 kB/s	29	179	15.72 Mbps	14.52 Mbps
k8s-worker-8	k8s-worker-8	9.66 hour	15.51 GiB	4	5.52	45.18%	40.22%	41.54%	9.21 kB/s	15.20 kB/s	32	177	8.38 Mbps	7.83 Mbps
k8s-worker-9	k8s-worker-9	9.66 hour	15.51 GiB	6	3.80	35.02%	43.22%	29.95%	28.55 kB/s	22.80 kB/s	29	25	24.63 Mbps	25.67 Mbps

看上图就可以知道，有几个主机的带宽都飘红了，而其他的资源使用率并没有特别高。前面我们有说过，分析网络问题，不应该只看网络带宽，还要分析其他的内容，下面我们就得分析一下网络带宽。

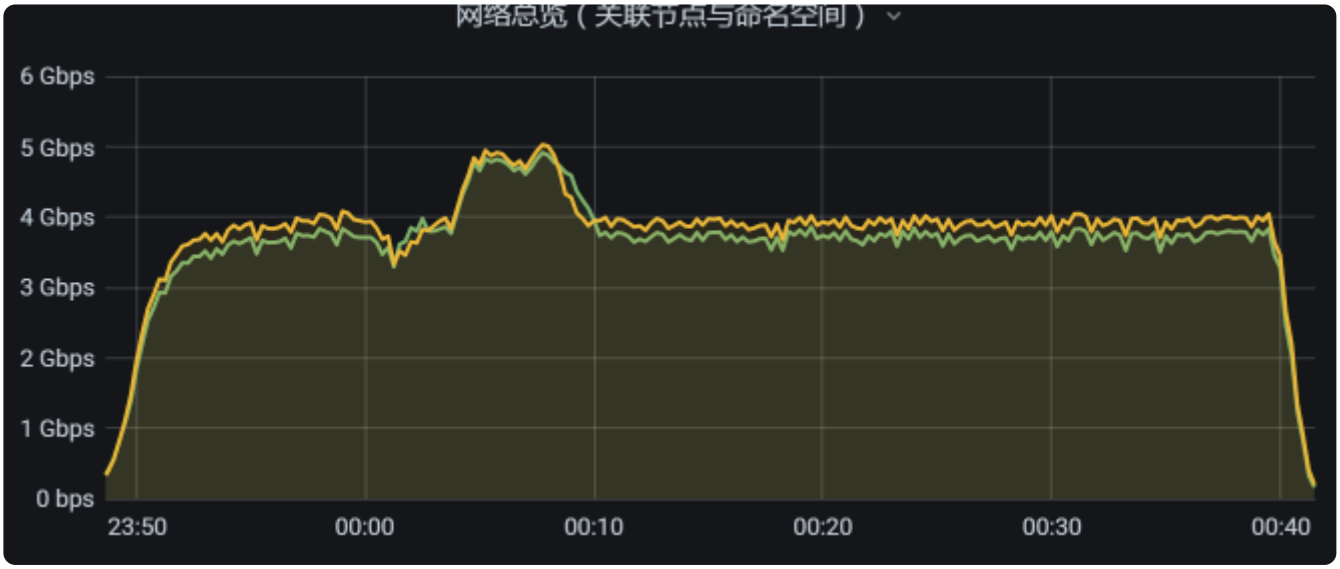
我们到监控工具中看一下网络的流量，你可以看到确实有一些非被测应用在占用带宽，并且占得还不小：



Current Status

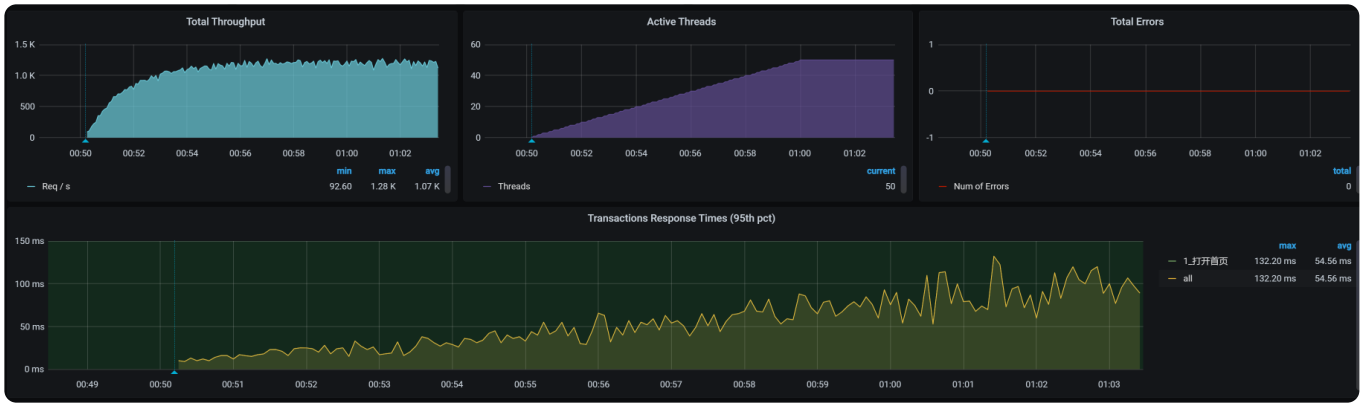
Pod	Bandwidth Received	Bandwidth Transmitted	Rate of Received Packets	Rate of Transmitted Packets	Rate of Received Packets Dropped	Rate of Transmitted Packets Dropp
weave-scope-agent-cmpvn	32.64 MB/s	38.92 MB/s	49.18 kpps	46.74 kpps	0 pps	0 pps
weave-scope-agent-dggxv	41.81 MB/s	38.39 MB/s	48.51 kpps	50.09 kpps	0 pps	0 pps
weave-scope-agent-snbxs	18.08 MB/s	19.87 MB/s	21.04 kpps	22.57 kpps	0 pps	0 pps
weave-scope-agent-6556c	6.43 MB/s	6.54 MB/s	2.56 kpps	2.57 kpps	0 pps	0 pps
weave-scope-agent-qj4h9	5.89 MB/s	5.69 MB/s	2.72 kpps	2.59 kpps	0 pps	0 pps
weave-scope-agent-2sg2h	4.10 MB/s	4.04 MB/s	2.09 kpps	2.02 kpps	0 pps	0 pps
weave-scope-agent-bz9qx	2.62 MB/s	3.70 MB/s	2.00 kpps	1.34 kpps	0 pps	0 pps
weave-scope-agent-ghf4k	2.22 MB/s	2.73 MB/s	1.83 kpps	1.37 kpps	0 pps	0 pps

我们再看总体带宽，发现已经用了 4G 多：



为了弄清楚那些与被测系统无关的应用，会对带宽消耗产生影响进而影响 TPS，我们现在先把影响带宽的应用都删除了，比如 Weave Scope、Monitoring 的监控工具等，从列表中来看这些应用占了不小的带宽。

然后我们再次测试，发现 TPS 有所上升，关键是稳定了很多：



我们可以看到，TPS 已经上升到了 1200 左右，可见带宽对 TPS 还是造成了不小的影响。

接着，我们查一下网络的队列，发现应用上面已经出现了不小的 Recv_Q。

复制代码

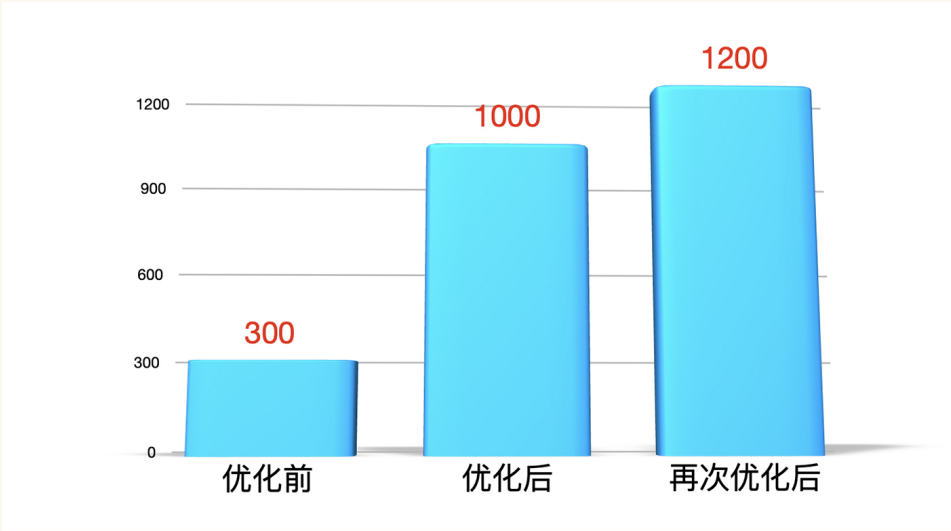
```
1 Proto Recv-Q Send-Q Local Address           Foreign Address         State
2 tcp       759      0 10.100.69.229:8085     10.100.140.32:35444     ESTABLISHE
3 tcp       832      0 10.100.69.229:34982    10.96.224.111:3306      ESTABLISHE
4 tcp      1056      0 10.100.69.229:34766    10.96.224.111:3306      ESTABLISHE
5 tcp       832      0 10.100.69.229:35014    10.96.224.111:3306      ESTABLISHE
```

```
 6 tcp      3408      0 10.100.69.229:34912     10.96.224.111:3306      ESTABLISHE
 7 tcp      3408      0 10.100.69.229:35386     10.96.224.111:3306      ESTABLISHE
 8 tcp      3392      0 10.100.69.229:33878     10.96.224.111:3306      ESTABLISHE
 9 tcp       560      0 10.100.69.229:35048     10.96.224.111:3306      ESTABLISHE
10 tcp      1664      0 10.100.69.229:34938     10.96.224.111:3306      ESTABLISHE
11 tcp       759      0 10.100.69.229:8085      10.100.140.32:35500     ESTABLISHE
12 tcp       832      0 10.100.69.229:35114     10.96.224.111:3306      ESTABLISHE
13 tcp      1056      0 10.100.69.229:34840     10.96.224.111:3306      ESTABLISHE
14 tcp      1056      0 10.100.69.229:35670     10.96.224.111:3306      ESTABLISHE
15
```

从这里来看，网络已经成为了下一个瓶颈（关于这一点，我们在后续的课程里会讲）。

如果你想接着调优，还可以从应用代码下手，让应用处理得更快。不过，对于基准测试来说，一个没有走任何缓存的接口，在一个 6C16G 的单节点虚拟机上能达到这么高的 TPS，我觉得差不多了。

接下来，我们还要去折腾其他的接口，所以，我们对这个接口的优化到这里就结束了。



总结

在打开首页这个接口的基准场景中，涉及到了很多方面的内容。从一开始的信息整理，比如访问路径、查看代码逻辑、场景试运行等，都是在为后面的分析做准备。

而当我们看到响应时间高，然后做拆分时间这一步，就是我一直在 RESAR 性能工程中强调的“**分析的起点**”。因为在此之前，我们用的都是压力工具上的数据，只是把它们罗列出

来就好了，没有任何分析的部分。

对于拆分时间，我们能用的手段有多种，你可以用你喜欢的方式，像日志、APM 工具，甚至抓包都是可以的。拆分了时间之后，我们就要分析在某个节点上响应时间高的时候，要怎么做。这时就用到了我一直强调的“全局 - 定向”监控分析思路。

在每一个阶段，你一定要清楚地定义优化的方向和目标，否则容易迷失方向。特别是对于一些喜欢把鼠标操作得特别快的同学，容易失去焦点，我劝你慢点操作，想清楚下一步再动。

而我们上述整个过程，都依赖于我说的性能分析决策树。从树顶往下，一层层找下去，不慌不乱，不急不燥。

只要你想，就能做到。

课后作业

最后，我给你留三个思考题。

1. 当 st cpu 高的时候，你要去看什么？
2. 当 wa cpu 高的时候，你要去看什么？
3. 为什么我们要把硬件资源用完？

记得在留言区和我讨论、交流你的想法，每一次思考都会让你更进一步。

如果你读完这篇文章有所收获，也欢迎你分享给你的朋友，共同学习进步。我们下一讲再见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 11 | 打开首页之一：一个案例，带你搞懂基础硬件设施的性能问题

下一篇 13 | 用户登录：怎么判断线程中的Block原因?

精选留言 (5)

写留言

微思

2021-04-17

优化过程很过瘾，希望老师能把实战课的案例部署文档整一个分享一下，也让我们本地能弄一个差不多一样的环境跟着课程实操一把，那就更过瘾啦！！！

展开

作者回复: 正在写部署文档，打算加餐中放。还在验证文档中。

3

微思

2021-04-17

1、当 st cpu 高的时候，你要去看什么?
-- 看宿主机的CPU运行模式
2、当 wa cpu 高的时候，你要去看什么?
-- cat /proc/interrupts cat /proc/softirqs 找到对应的模块和模块的运行原理，最后找出解决方案...

展开

作者回复: 一般毛刺不多，只要对整体趋势影响不大，我不会特意去分析。
在这个示例中，我没有分析，不知道是什么原因引起的。 如果想分析这样的毛刺也简单，思路一样。
请注意，不要看到现象直接问原因，不分析是不可能知道原因的。

3

同心飞翔

2021-04-16

老师，计算次数哪里为啥使用的43056而不是上面util100%对应的写19968？这里没看懂
另外，网络使用情况那个图，使用什么命令看的？谢谢

展开

作者回复: 这里要用字节数来算，不能用使用率来算。
网络的是iftop命令。

1

门窗小二
2021-04-22

老师可以把从哪个参数看出是顺序写，就像@同心飞翔说的那样为啥不用第一行io的数据19968而用第二行的数据43056；还有一点希望在图片标示出来接下来要分析使用到的数据

展开

作者回复: 因为数据是用代码的形式放在课程中的，所以无法加上不同的颜色标示。也许编辑小姑娘能有招吧。
在下面的计算过程中，我不是写了计算公式吗？如果一次写了多个block就显然是顺序写了；如果一次只写一个block就显然是随机写了。

1

Geek_bbe4c8
2021-04-17

“在网络中当小包过多的时候，网络带宽是难以达到线性流量的。”
老师，这里的线性流量怎么理解？

展开

作者回复: 简单来说，就是假设一个包是1M，那总带宽是100M的话，需要100个包就行。
如果一个包只有10bit，那就不可能用完100M的总带宽。